System design: Google Docs

Suresh Podeti
8 min read · Oct 26, 2023

Introduction

Google docs is a collaborative document editing service.

A collaborative document editing service can be designed in two ways:

- It could be designed as a centralized facility using client-server architecture to provide document editing service to all users.
- It could be designed using peer-to-peer technology to collaborate on a single document.

Most commercial solutions focus on client-service architecture to have finer control. Therefore, we'll focus on designing a service using the client service architecture. Let's see how we will progress in this chapter.

Requirements

Functional

- **Document collaboration**: Multiple users should be able to edit a document simultaneously. Also, a large number of users should be able to view a document.
- **Conflict resolution**: The system should push the edits done by one user to all the other collaborators. The system should also resolve conflicts between users if they're editing the same portion of the document.
- **Suggestions**: The user should get suggestions about completing frequently used words, phrases, and keywords in a document, as well as suggestions about fixing grammatical mistakes.
- **View count**: Editors of the document should be able to see the view count of the document.
- **History**: The user should be able to see the history of collaboration on the document.

Non-functional

- **Latency**: Different users can be connected to collaborate on the same document. Maintaining low latency is challenging for users connected from different regions.
- **Consistency**: The system should be able to resolve conflicts between users editing the document concurrently, thereby enabling a consistent view of the document. At the same time, users in different regions should see the updated state of the document. Maintaining consistency is important for users connected to both the same and different zones.
- **Availability**: The service should be available at all times and show robustness against failures.
- **Scalability**: A large number of users should be able to use the service at the same time. They can either view the same document or create new documents.

Components

Data stores

- **Relational database —** for saving users' information and document-related information for imposing privilege restrictions
- **NOSQL —** for storing user comments for quicker access.
- **Time series —** to save the edit history of documents
- **Blob storage —** to store videos and images within a document

- **Cache** — distributed cache like **Redis** and a **CDN** to provide end users good performance. We use Redis specifically to store different data structures, including user sessions, features for the typeahead service, and frequently accessed documents. The CDN stores frequently accessed documents and heavy objects, like images and videos.

Processing queue

Using an HTTP call for sending every minor character is inefficient. Therefore, we'll use **WebSockets** to reduce overhead and observe live changes to the document by different users.

Other components

Other components include the session servers that maintain the user's session information. We'll manage document access privileges through the session servers. Essentially, there will also be **configuration, monitoring, pub-sub, and logging services** that will handle tasks like monitoring and electing leaders in case of server failures, queueing tasks like user notifications, and logging debugging information.

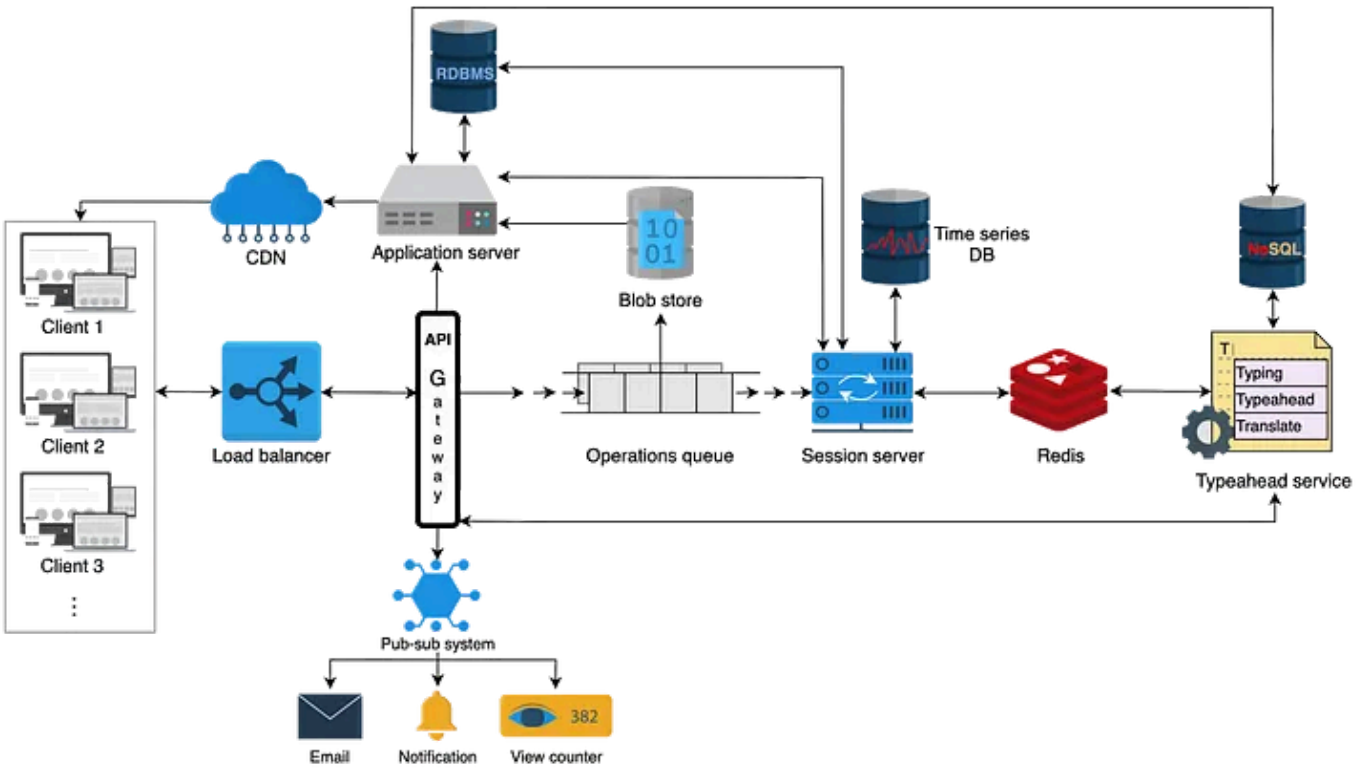


Fig 1.0: A detailed design of the collaborative document editing service

Workflow

- **Collaborative editing and conflict resolution**: Each request gets forwarded to the operations queue. This is where conflicts get resolved between different collaborators of the same document. If there are no conflicts, the data is batched and stored in the **time series database** via session servers. Data like videos and images get compressed for storage optimization, while characters are processed right away.

- **History**: It's possible to recover different versions of the document with the help of a time series database. Different versions can be compared using DIFF operations that compare the versions and identify the differences to recover older versions of the same document.

- **Asynchronous operations**: Notifications, emails, view counts, and comments are asynchronous operations that can be queued through a **pub-sub component like Kafka**. The API gateway generates these requests and forwards them to the pub-sub module

- **Suggestions**: Suggestions are in the form of the typeahead service that offers autocomplete suggestions for typically used words and phrases. The typeahead service can also extract attributes and keywords from within the document and provide suggestions to the user. Since the number of words can be high, we'll use a NoSQL database for this purpose. In addition, most frequently used words and phrases will be stored in a caching system like Redis

- **Import and export documents**: The application servers perform a number of important tasks, including importing and exporting documents. Application servers also convert documents from one format to another. For example, a `.doc` or `.docx`

document can be converted in to `.pdf` or vice versa. Application servers are also responsible for feature extraction for the typeahead service.

Detailed design

Document editor

A **document** is a composition of **characters** in a specific order. Each character has a **value and a positional index**. The character can be a letter, a number, an enter (
), or a space. An index represents the character's position within the ordered list of characters.

The job of the text or document editor is to perform operations like `insert()`, `delete()`, `edit ()`, and more on the characters within the document. A depiction of a document and how the editor will perform these operations is below.

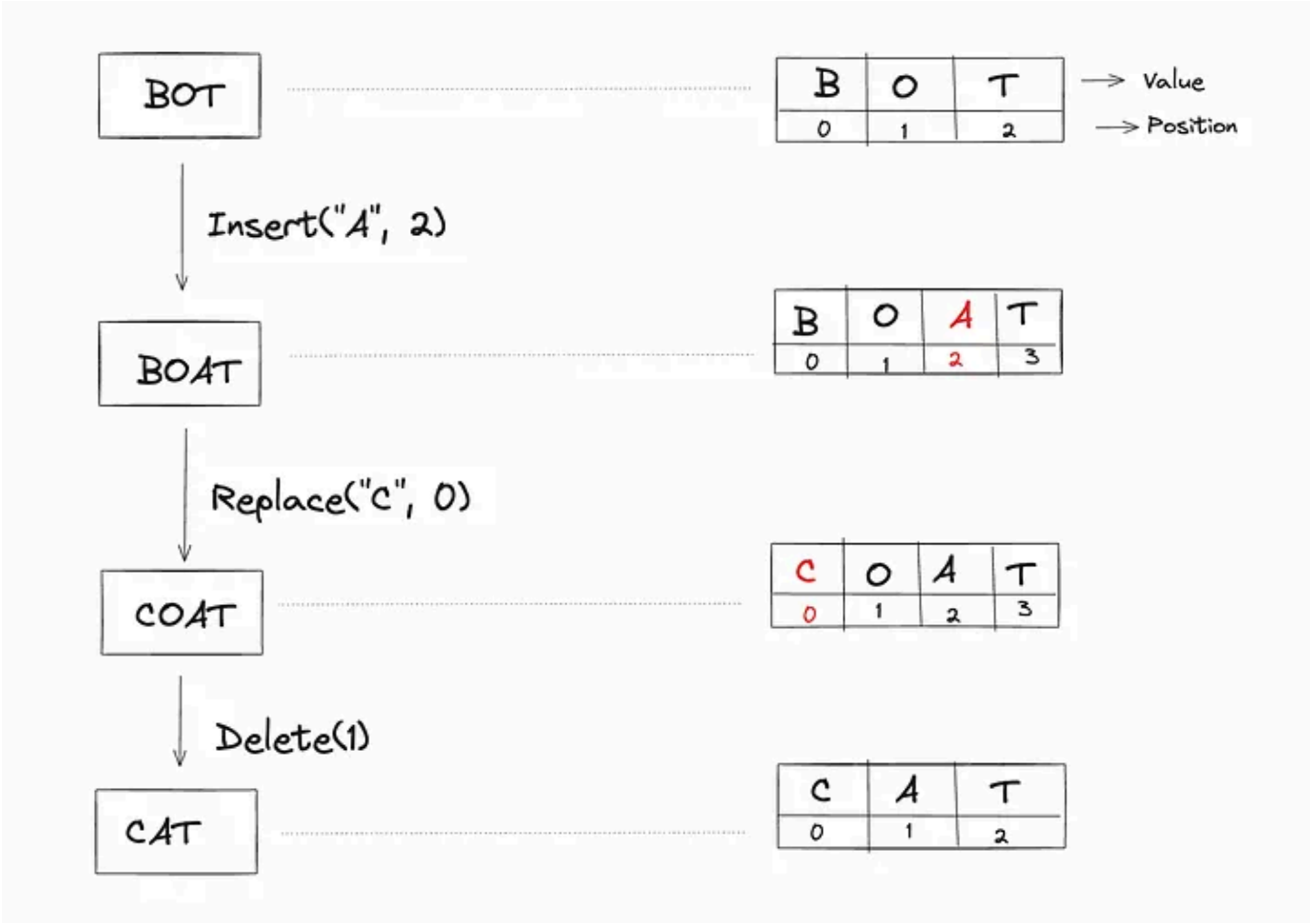


Fig 2.0: How a document editor perform various operations

Concurrency

Collaboration on the same document by different users can lead to concurrency issues. Conflicts may arise whenever multiple users edit the same portion of a document. Since users have a local copy of the document, the final status of the document may be different at the server from what the users see at their end. After the server pushes the updated version, users discover the unexpected outcome.

Adding character at the same positional index

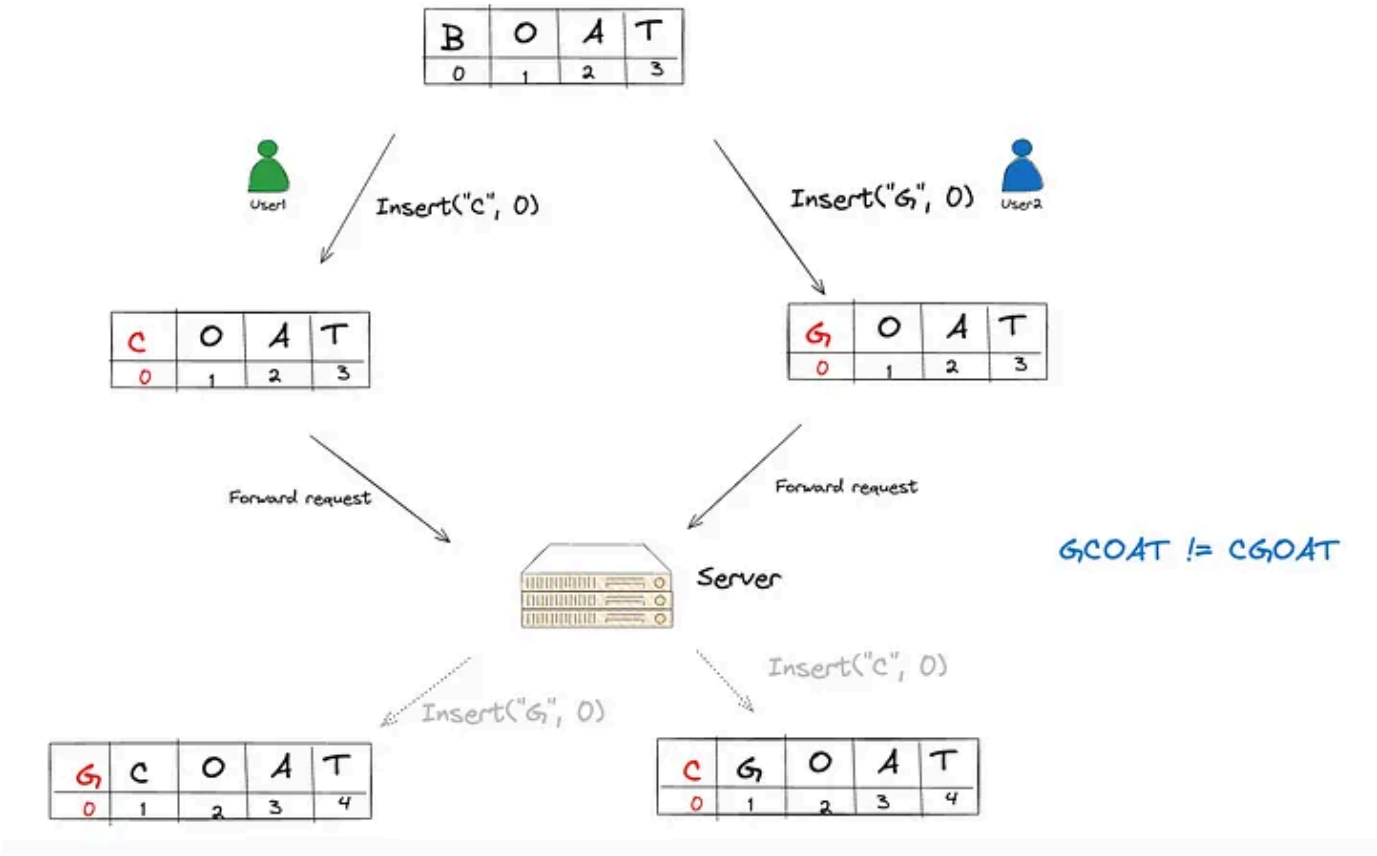


Fig 3.0: How two users modifying the same character can lead to concurrency issues

Delete the same character

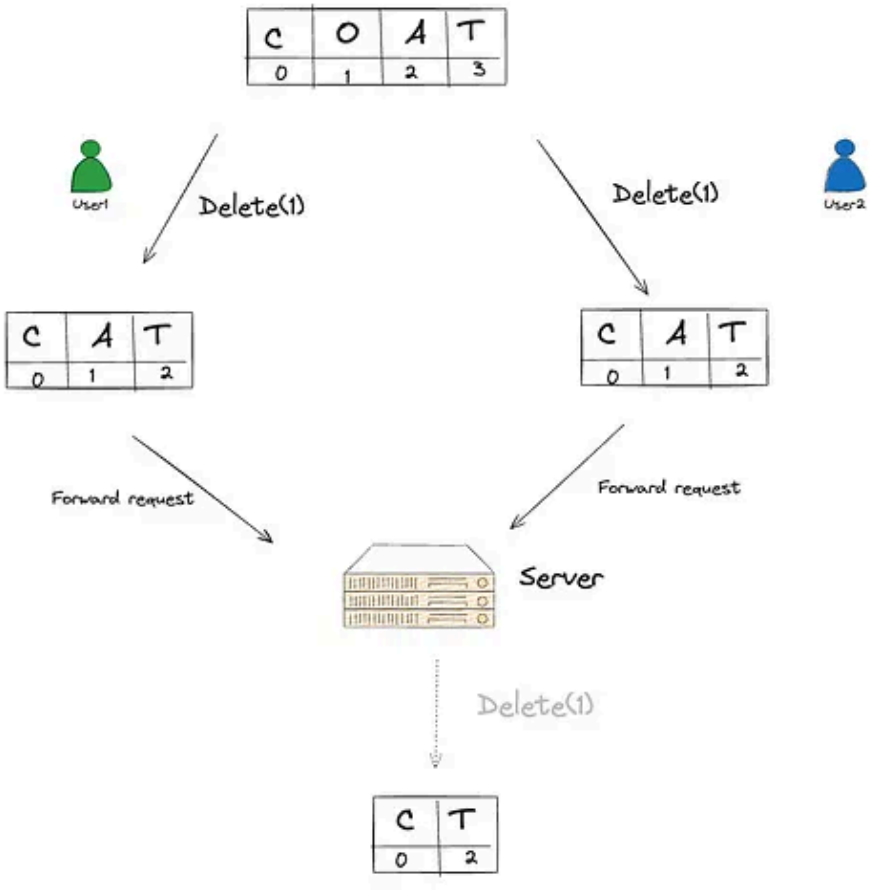


Fig 4.0: How deleting the same character can lead to unexpected changes

This second example shows that different users applying the same operation **won't be idempotent**. So, conflict resolution is necessary where multiple collaborators are editing the same portion of the document at the same time.

Solution to concurrency issues in collaborative editing should respect two rules:

- **Commutativity**: The order of applied operations shouldn't affect the end result.
- **Idempotency**: Similar operations that have been repeated should apply only once.

Techniques for conflict resolution

Operational transformation(OT)

Operational transformation (OT) is a technique that's widely used for conflict resolution in collaborative editing. It's a **lock-free** and **non-blocking approach** for conflict resolution. If operations between collaborators conflict, OT resolves conflicts and pushes the correct converged state to end users. As a result, OT provides consistency for users.

OT performs operations using the **positional index method** to resolve conflicts, such as the ones we discussed above. OT resolves the problems above by holding commutativity and idempotency.

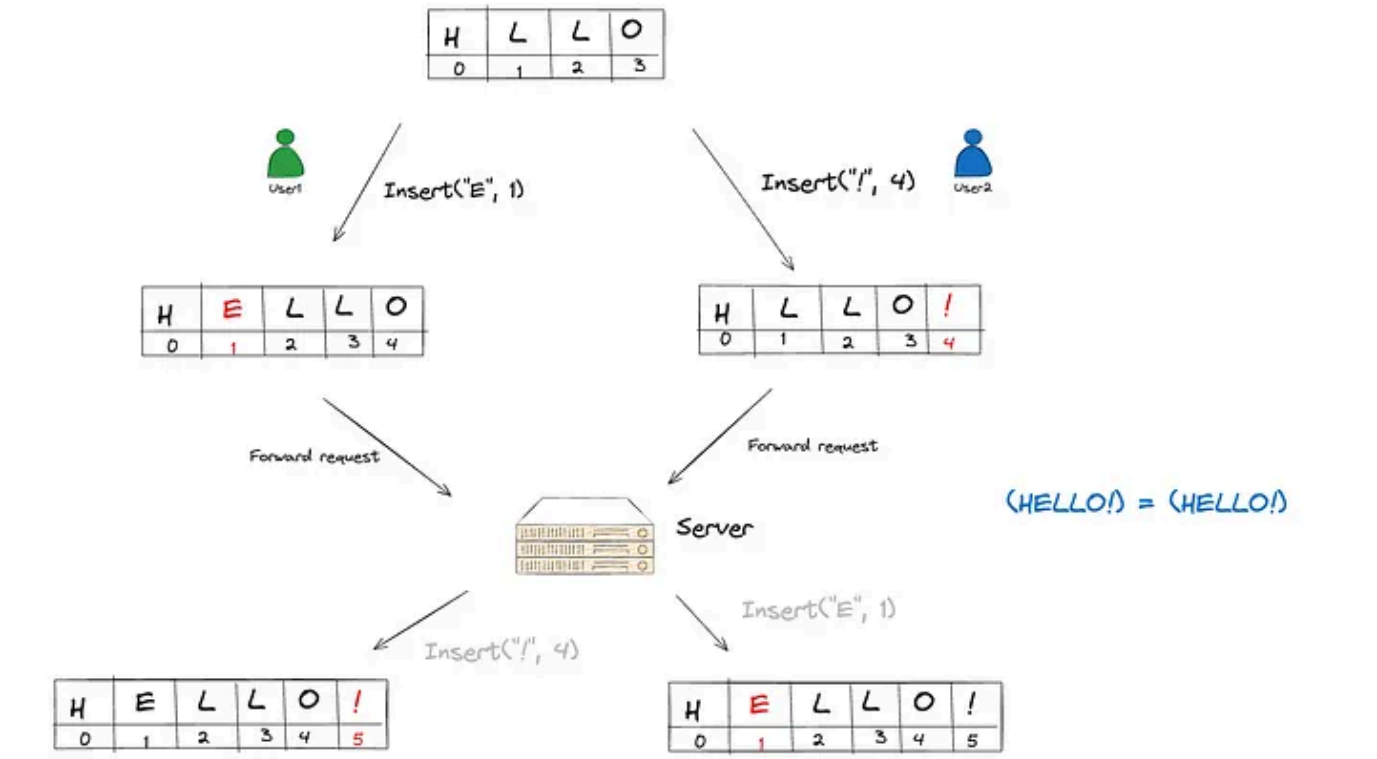


Fig 5.0: Operational transformation working example

Collaborative editors based on OT are consistent if they have the following two properties:

- **Causality preservation**: If operation `a` happened before operation `b`, then operation `a` is executed before operation `b`.
- **Convergence**: All document replicas at different clients will eventually be identical.

The properties above are a part of the **CC consistency model**, which is a model for consistency maintenance in collaborative editing.

OT has two disadvantages:

- Each operation to characters may require changes to the positional index. This means that operations are **order dependent** on each other.
- It's challenging to develop and implement.

Operational transformation is a **set of complex algorithms**, and its correct implementation has proved challenging for real-world applications. For example, the Google Wave team t**ook two years to implement** an OT algorithm.

Conflict-free Replicated Data Type (CRDT)

The **Conflict-free Replicated Data Type (CRDT)** was developed in an effort to improve OT. A CRDT has a complex data structure but a simplified algorithm.

A CRDT satisfies both commutativity and idempotency by assigning two key properties to each character:

- It **assigns a globally unique identity** to each character.
- It **globally orders** each character.

Each operation now has an updated data structure:

Data	Explanation	Example
SiteID	Unique site identifier in the form of a UUID	56813cf4-66d2-4d68-85eb-684978cd9d29
Value	Value of character	"A"
PositionalIndex	Unique position of each character	4.5

Fig 6.0: Simplified data structure of a CRDT

The `SiteID` uniquely identifies a user's site requesting an operation with a `Value` and a `PositionalIndex`. The value of `PositionalIndex` can be in fractions for two main reasons.

- The `PositionalIndex` of other characters won't change because of certain operations.
- The order dependency of operations between different users will be avoided.

The example below depicts that a user from site ID `123e4567-e89b-12d3` is inserting a character with a value of `A` at a `PositionalIndex` of `1.5`. Although a new character is added, the positional indexes of existing characters are preserved using fractional indices. Therefore, the order dependency between operations is avoided. As shown below, an `insert()` between `O` and `T` didn't affect the position of `T`.

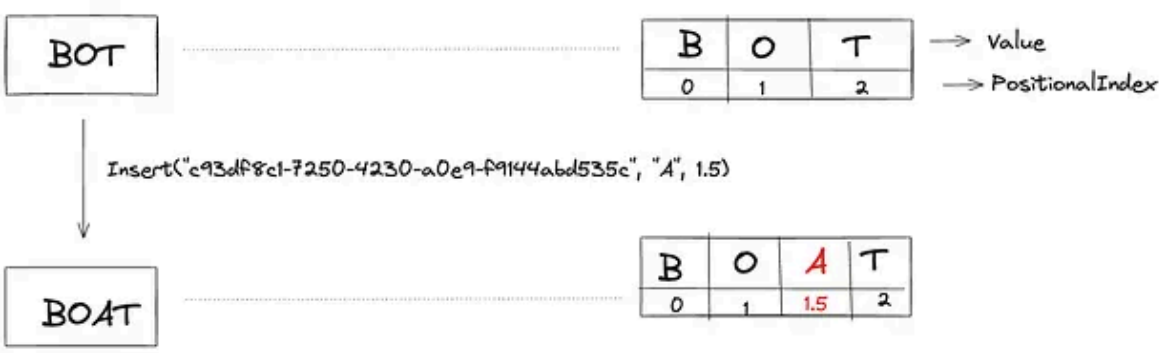


Fig 7.0: Preventing order dependency between operations

CRDTs ensure strong consistency between users. Even if some users are offline, the local replicas at end users will converge when they come back online.

Although well-known online editing platforms like Google Docs, Etherpad, and Firepad use OT, CRDTs have made concurrency and consistency in collaborative document editing easy. In fact, with CRDTs, it's possible to implement a serverless peer-to-peer collaborative document editing service.

Note: OT and CRDTs are good solutions for conflict resolution in collaborative editing, but our use of WebSockets makes it possible to highlight a collaborator's cursor. Other users will anticipate the position of a collaborator's next operation and naturally avoid conflicts.

Evaluation

Consistency

We've looked at how we'll achieve strong consistency for conflict resolution in a document through two technologies: operational transformation (OT) and Conflict-free Resolution Data Types (CRDTs). In addition, a time series database enables us to preserve the order of events. Once OT or CRDT has resolved any conflicts, the final result is saved in the database. This helps us achieve consistency in terms of individual operations.

We're also interested in keeping the document state consistent across different servers in a data center. To replicate an updated state of a document within the same data center at the same time, we can use peer-to-peer protocols like Gossip protocol. Not only will this strategy improve consistency, it will also improve availability.

Availability

Our design ensures availability by using replicas and monitoring the primary and replica servers using monitoring services. Key components like the operations queue and data stores internally manage their replication.

Since we use WebSockets, our WebSocket servers can connect users to the session maintenance servers that will determine if a user is actively viewing or collaborating on a document. Therefore, keeping multiple WebSocket servers will increase the availability of the design. Lastly, we employ caching services and CDNs to improve availability in case of failures.

Scalability

Since we've used microservice architecture, we can easily scale each component individually in case the number of requests on the operations queue exceeds its capacity. We can use multiple operations queues. In that case, each operations queue will be responsible for a single document. We can forward operations requested by different users that are associated with a single document to a specific queue. The number of spawned queues will be equal to the number of active documents. As a result, we're able to achieve horizontal scalability.

Google Docs | System Design Interview | Software Architect | Software Architecture | Distributed System Design

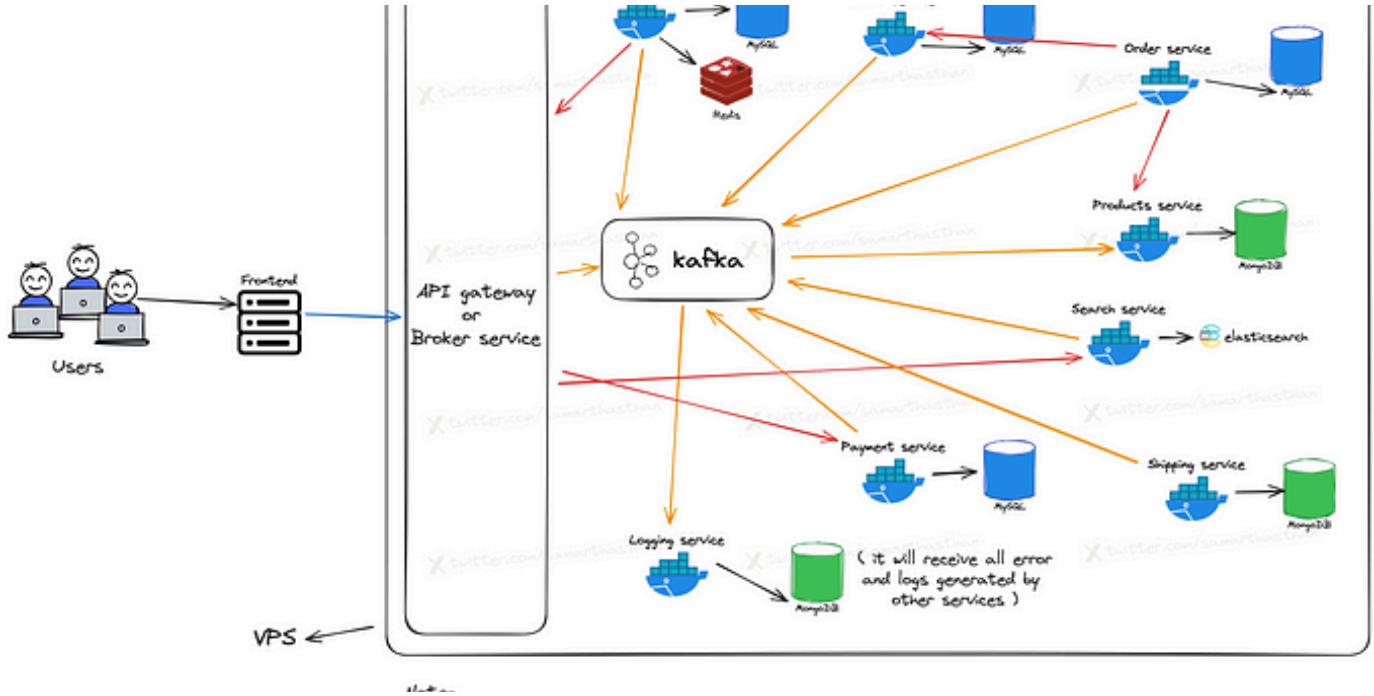
Edit profile

Written by Suresh Podeti

1.2K Followers

Aspiring Software Architect | SDE III (Golang) @JungleeGames| Ex-Byju's | Co-founder @Rupant Tech. | Ex-Synopsys | I.I.T Kharagpur | 7+ Years of Experience

Recommended from Medium



Samarth Asthan

Building a Scalable E-commerce Empire: A Micro-services System Design Approach

Hey everyone! I'm Samarth Asthan, a 3rd-year computer science student, and I'm new to the world of system design. But, I'm excited to share...

3 min read · Dec 11, 2023

8